REMARKS

Summary of the Office Action

Claims 1-3, 7-14, 16-19, 23-26, 30-37 and 39 are considered in the Office Action.

Claims 1-3, 7-14, 16-19, 23-26, 30-37 and 39 have been rejected under 35 U.S.C. § 112, second paragraph, as being indefinite for failing to particularly point out and distinctly claim the subject matter which applicant regards as the invention.

Claims 1-2, 17-18 and 24-25 have been rejected under 35 U.S.C. § 103(a) as being obvious over Bloomquist U.S. Patent 6,295,133 ("Bloomquist").

Claims 3, 8-9, 12-14, 19, 26, 31-32 and 35-37 have been rejected to under 35 U.S.C. § 103(a) as obvious over Bloomquist in view of Gass U.S. Patent No. 5,822,503 ("Gass").

Claims 7, 23 and 30 are rejected under 35 U.S.C. § 103(a) as obvious over Bloomquist in view of Hains U.S. Patent No. 6,262,811 ("Hains").

Claims 16 and 39 are rejected to under 35 U.S.C. § 103(a) as obvious over Bloomquist in view of Gass and Hains.

Claims 10-11 and 33-34 are rejected to under 35 U.S.C. § 103(a) as obvious over Bloomquist in view of Gass and Ng U.S. Patent No. 6,131, 096 ("Ng").

Summary of the Reply

Applicant has amended claims 1-3, 17-19 and 24-26, and has cancelled claims 8-14, 16, 31-37 and 39 without prejudice. Applicant respectfully submits that the cited references do not describe or suggest the amended claims.

Reply to § 112, Second Paragraph Rejections

Claims 1-3, 7-14, 16-19, 23-26, 30-37 and 39 have been rejected under § 112, second paragraph as being indefinite. In particular, the Office action states that claims 1, 8 10, 14, 17, 24, 31 33 and 37 each specifically recite "a PostScript pattern," but that PostScript is not a clearly and particularly defined page description language, and therefore the claims are indefinite.

Applicant respectfully disagrees that the claims are indefinite because the claims recite "PostScript." Indeed, applicant finds this rejection quite odd, because as of today's date, the U.S. Patent & Trademark Office ("PTO") has issued 155 patents[1] that recite "PostScript" in their claims, including patents whose claims expressly reference the PostScript language or PostScript page description language.

In addition, whereas on page 4 of the Office action, PostScript is described as "not a clearly and particularly defined [page description language]," on the very next page the Office action states that PostScript "is a well-known, typical and easily accessible for [sic] of Page Description Language." Applicant respectfully submits that these two statements are inconsistent and cannot be reconciled, and further undercut any merit of the § 112, second paragraph rejections.

Because the claim language is not indefinite, applicant respectfully requests that the Examiner withdraw the § 112, second paragraph rejections.

---

[1] U.S. Patent Nos.: 4,011,545, 4,533,993, 4,931,927, 4,965,763, 5,029,115, 5,075,874, 5,187,775, 5,222,200, 5,239,621, 5,305,020, 5,392,419, 5,446,896, 5,455,599, 5,473,741, 5,502,637, 5,553,200, 5,555,435, 5,559,933, 5,570,459, 5,603,043, 5,617,528, 5,619,624, 5,659,638, 5,671,345, 5,713,032, 5,748,860, 5,797,320, 5,809,218, 5,857,209, 5,859,958, 5,862,270, 5,892,900, 5,923,821, 5,953,392, 5,963,966, 5,978,477, 5,982,727, 5,998,609, 5,999,709, 6,006,281, 6,009,462, 6,020,970, 6,031,628, 6,049,339, 6,052,198, 6,057,858, 6,057,930, 6,061,057, 6,067,406, 6,070,175, 6,078,403, 6,092,089, 6,111,654, 6,115,508, 6,124,858, 6,141,006, 6,192,157, 6,199,073, 6,205,452, 6,208,431, 6,215,502, 6,226,776, 6,229,623, 6,247,011, 6,252,671, 6,262,806, 6,268,927, 6,289,364, 6,295,134, 6,317,848, 6,323,864, 6,327,050, 6,331,895, 6,332,150, 6,347,852, 6,353,483, 6,360,236, 6,362,895, 6,366,650, 6,370,602, 6,381,032, 6,385,350, 6,393,442, 6,396,593, 6,401,141, 6,404,431, 6,411,314, 6,424,430, 6,429,947, 6,429,950, 6,433,811, 6,456,395, 6,462,756, 6,483,524, 6,507,848, 6,510,459, 6,529,214, 6,542,252, 6,547,831, 6,587,972, 6,611,349, 6,615,212, 6,615,372, 6,631,375, 6,633,890, 6,647,437, 6,657,740, 6,661,919, 6,662,270, 6,690,489, 6,701,023, 6,717,691, 6,721,769, 6,726,104, 6,728,419, 6,738,155, 6,738,951, 6,778,684, 6,789,234, 6,791,707, 6,801,935, 6,816,904, 6,825,864, 6,829,707, 6,833,932, 6,839,149, 6,850,946, 6,856,428, 6,897,974, 6,898,601, 6,899,475, 6,903,839, 6,912,294, 6,914,533, 6,914,893, 6,915,484, 6,938,051, 6,943,809, 6,952,780, 6,957,235, 6,961,900, 6,963,668, 6,990,904, 6,992,785, 6,992,790, 6,996,351, 7,002,700, 7,003,723, 7,036,076, 7,042,666, 7,057,370, 7,069,314, 7,072,067, 7,092,117, 7,107,522.

Reply to § 103(a) Rejections

This application describes and claims methods and apparatus that may be used to proof a print job that requires specialized print media, such as textiles, fabrics, wood, metals, and other similar specialized print media, on a paper-based printer. In particular, the invented methods and apparatus associate a spot color name (e.g., "Gold Foil") with a corresponding image (e.g., an image that looks like gold foil). For example, a RIP may include a database that includes spot color names associated with corresponding images. A user may then create a print job that includes one or more of the database spot color names. For example, a user may use Adobe Illustrator to create a document that includes a background image that is painted with the "Gold Foil" spot color. The user may then use the application program to generate a PostScript print job that includes a reference to the "Gold Foil" spot color, and may submit the print job to the RIP for printing. When the RIP interprets the print job, it will look for any occurrence of database spot color names (such as "Gold Foil") in the print job. If a database spot color name is found, the RIP will retrieve the corresponding image from the database, and then insert PostScript pattern code into the print job to "paint" the retrieved image in the print job.

Accordingly, amended independent claims 1, 17 and 24 recite methods, apparatus and corresponding program storage media that: (1) provide a database including a spot color name associated with corresponding image data; (2) receive a print job including PDL code that includes a reference to the spot color name; (3) identify the spot color name in the PDL code; (4) retrieve from the database the corresponding image data associated with the identified spot color name; (5) add PDL code to the print job for painting the retrieved image data as a PostScript pattern in the print job; (6) execute the PDL code in the print job; and (7) paint the retrieved image data as a PostScript pattern in the print job.

As explained in the PostScript Language Reference, 3rd Edition (excerpt attached hereto as Exhibit A), "PostScript patterns" include tiling patterns, which consist of a small graphical figure that is replicated at fixed horizontal and vertical intervals to fill an area to be painted, and shading patterns, which define a gradient fill that produces a smooth transition between colors across the area to be painted. PostScript patterns are specified in a special color space family named Pattern, whose

"color values" are pattern dictionaries that contain descriptive information defining the appearance and properties of a pattern. Further, PostScript includes specific makepattern and setpattern operators that are used to paint with a pattern.

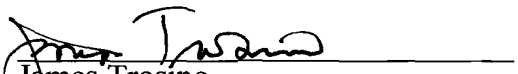
None of the cited references describe or suggest anything regarding PostScript patterns. Indeed, various comments in the Office action indicate that the Examiner has confused the term "PostScript pattern" with "PostScript." First, the Office action at 3 states that "PostScript patterns are taught in the background of Bloomquist as a typical Page Description Language format from which output is derived" (citing Bloomquist Col. 1, lines 63-66). The cited portion, however, does not state anything regarding PostScript patterns, but merely states that imaging application software typically provides output in the format of a PDL, such as PostScript. Second, the Office action at 3 states that Gass "also teaches PostScript patterns, particularly Encapsulated PostScript patterns." Although Gass describes a method of modifying one or more colors contained in an encapsulated PostScript file, Gass does not describe anything regarding PostScript patterns. Finally the Office action at 3 states that "PostScript patterns are well known and expected in the art. The PostScript PDL is simply one of many proprietary types of PDL." These two sentences do not make any sense, unless the Examiner has interpreted "PostScript patterns" as "PostScript."

The term "PostScript pattern" is not synonymous with "PostScript." References (such as Bloomquist and Gass) that only pertain to some aspect of PostScript, but that do not describe anything regarding PostScript patterns do not anticipate or render obvious the claimed invention. None of the cited references describe or suggest the claimed invention. Accordingly, applicant respectfully requests that the Examiner withdraw the § 103 rejections.

Conclusion

For the reasons stated above, applicant submits that this application, including claims 1-3, 7, 17-19, 23-26 and 30, is allowable. Applicant therefore respectfully requests that the Examiner allow this application.

Respectfully submitted,



James Trosino
Registration No. 39,862
Attorney for Applicant

**EXHIBIT A**

# PostScript®
# LANGUAGE REFERENCE
# third edition

**Adobe Systems Incorporated**

**Addison-Wesley Publishing Company**
Reading, Massachusetts • Menlo Park, California • New York • Don Mills, Ontario
Harlow, England • Amsterdam • Bonn • Sydney • Singapore • Tokyo
Madrid • San Juan • Paris • Seoul • Milan • Mexico City • Taipei

Library of Congress Cataloging-in-Publication Data
PostScript language reference manual / Adobe Systems Incorporated. — 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-37922-8
1. PostScript (Computer program language) I. Adobe Systems.
QA76.73.P67 P67 1999
005.13'3—dc21 98-55489
CIP

© 1985–1999 Adobe Systems Incorporated. All rights reserved.

NOTICE: All information contained herein is the property of Adobe Systems Incorporated.

No part of this publication (whether in hardcopy or electronic form) may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the publisher.

PostScript is a registered trademark of Adobe Systems Incorporated. All instances of the name PostScript in the text are references to the PostScript language as defined by Adobe Systems Incorporated unless otherwise stated. The name PostScript also is used as a product trademark for Adobe Systems' implementation of the PostScript language interpreter.

Except as otherwise stated, any mention of a "PostScript printer," "PostScript software," or similar item refers to a product that contains PostScript technology created or licensed by Adobe Systems Incorporated, not to one that purports to be merely compatible.

Adobe, Adobe Illustrator, Adobe Type Manager, Chameleon, Display PostScript, FrameMaker, Minion, Myriad, Photoshop, PostScript, PostScript 3, and the PostScript logo are trademarks of Adobe Systems Incorporated. LocalTalk, QuickDraw, and TrueType are trademarks and Mac OS is a registered trademark of Apple Computer, Inc. Helvetica and Times are registered trademarks of Linotype-Hell AG and/or its subsidiaries. Times New Roman is a trademark of The Monotype Corporation registered in the U.S. Patent and Trademark Office and may be registered in certain other jurisdictions. Unicode is a registered trademark of Unicode, Inc. PANTONE is a registered trademark and Hexachrome is a trademark of Pantone, Inc. Windows is a registered trademark of Microsoft Corporation. All other trademarks are the property of their respective owners.

This publication and the information herein are furnished AS IS, are subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied, or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes, and noninfringement of third-party rights.

ISBN 0-201-37922-8
1 2 3 4 5 6 7 8 9 CRS 03 02 01 00 99
First printing February 1999

"Process Color Model" on page 422). The overprint parameter is a boolean flag that determines how painting operations affect colorants other than those explicitly or implicitly specified by the current color space.

If the overprint flag is *false* (the default value), painting a color in any color space causes the corresponding areas of unspecified colorants to be erased (painted with a tint value of 0.0). The effect is that the color marked at any position on the page is whatever was painted there last; this is consistent with the normal opaque painting behavior of the Adobe imaging model.

If the overprint flag is *true* and the output device supports overprinting, no such erasing actions are performed; anything previously painted in other colorants is left undisturbed. Consequently, the color at a given position on the page may be a combined result of several painting operations in different colorants. The effect produced by such overprinting is device-dependent and is not defined by the PostScript language.

***Note:*** *Not all devices support overprinting. Furthermore, many LanguageLevel 2 implementations support it only when separations are being produced, not for composite output. If overprinting is not supported, the value of the overprint parameter is ignored.*

## 4.9 Patterns

When operators such as **fill**, **stroke**, and **show** paint an area of the page with the current color, they ordinarily apply a single color that covers the area uniformly. Sometimes, however, it is desirable to apply "paint" that consists of a repeating figure or a smoothly varying color gradient instead of a simple color. Such a repeating figure or smooth gradient is called a *pattern*. Patterns are quite general, and have many uses. They can be used to create various graphical textures, such as weaves, brick walls, sunbursts, and similar geometrical and chromatic effects.

PostScript patterns come in two varieties:

- *Tiling patterns* consist of a small graphical figure (called a *pattern cell*) that is replicated at fixed horizontal and vertical intervals to fill the area to be painted.
- *Shading patterns* define a *gradient fill* that produces a smooth transition between colors across the area.

***Note:*** *The ability to paint with patterns is a feature of LanguageLevels 2 (tiling patterns) and 3 (shading patterns). With some effort, it is possible to achieve a limited form of tiling patterns in LanguageLevel 1 by defining them as character glyphs in a special font and painting them repeatedly with the* **show** *operator. Another technique, defining patterns as halftone screens, is* not *recommended, because the effects produced are device-dependent.*

Patterns are specified in a special color space family named **Pattern**, whose "color values" are *pattern dictionaries* instead of the numeric component values used with other color spaces. This section describes **Pattern** color spaces and the specification of color values within them; see Section 4.8, "Color Spaces," for information about color spaces and color values in general.

### 4.9.1 Using Patterns

A pattern dictionary contains descriptive information defining the appearance and properties of a pattern. All pattern dictionaries contain an entry named **PatternType**, whose value identifies the kind of pattern the dictionary describes: type 1 denotes a tiling pattern, type 2 a shading pattern. The remaining contents of the dictionary depend on the pattern type, and are detailed below in the sections on each pattern type.

Painting with a pattern is a five-step procedure:

1. *Define the prototype pattern.* Create a pattern dictionary containing descriptive information about the pattern's appearance and other properties.
2. *Instantiate the pattern.* Pass the prototype pattern dictionary to the **makepattern** operator. This produces a copy of the dictionary representing an instance of the pattern that is locked to current user space. The copy may contain an optional additional entry, named **Implementation**, containing implementation-dependent information to be used by the interpreter in painting the pattern.
3. *Select a* ***Pattern*** *color space.* Use the **setcolorspace** operator to set the current color space to a **Pattern** space. The initial color value in this color space is a *null* object, which is treated as if it were a pattern dictionary representing an empty tiling pattern. Painting with this pattern produces no marks on the current page.

4. *Make the pattern the current color.* Invoke **setcolor** with the instantiated pattern dictionary from step 2 as an operand (and possibly other operands as well) to select the pattern as the current color.
5. *Invoke painting operators*, such as **fill**, **stroke**, **imagemask**, or **show**. All areas that normally would be painted with a uniform color will instead be filled with the selected pattern.

A convenience operator, **setpattern**, combines steps 3 and 4 above into a single operation: it takes a pattern dictionary as an operand, selects a **Pattern** color space, and sets the specified pattern as the current color. **setpattern** is the normal method for selecting patterns; in practice, it is rarely necessary to set the color space and color value separately. For purposes of exposition, however, the examples presented here will separate the two steps for maximum clarity.

### 4.9.2 Tiling Patterns

A tiling pattern consists of a small graphical figure called a *pattern cell.* Painting with the pattern replicates the cell at fixed horizontal and vertical intervals to fill an area. The effect is as if the figure were painted on the surface of a clear glass tile, identical copies of which were then laid down in an array covering the area and trimmed to its boundaries. This is called *tiling* the area.

The pattern cell can include graphical elements such as filled areas, text, and sampled images. Its shape need not be rectangular, and the spacing of tiles can differ from the dimensions of the cell itself. The cell's appearance is defined by an arbitrary PostScript procedure, the ***PaintProc*** *procedure*, which paints a single instance of the cell. The PostScript interpreter obtains the **PaintProc** procedure from the pattern dictionary and calls it (with the graphics state altered in certain ways) to obtain the pattern cell. When performing painting operations such as **fill** or **stroke**, the interpreter then paints the cell on the current page as many times as necessary to fill an area. To optimize execution, the interpreter maintains a cache of recently used pattern cells.

Tiling patterns are defined by pattern dictionaries of type 1. Table 4.9 lists the entries in this type of dictionary. (The dictionary can also contain any additional entries that its **PaintProc** procedure may require.) All entries except **Implementation** can appear in a prototype pattern dictionary supplied as an operand to **makepattern**. The pattern dictionary instantiated and returned by

**makepattern** may contain an **Implementation** entry in addition to those included in the prototype.

**TABLE 4.9 Entries in a type 1 pattern dictionary**

| KEY | TYPE | VALUE |
|---|---|---|
| **PatternType** | integer | *(Required)* A code identifying the pattern type that this dictionary describes; must be 1 for a tiling pattern. |
| **XUID** | array | *(Optional)* An *extended unique ID* that uniquely identifies the pattern (see Section 5.6.2, "Extended Unique ID Numbers"). The presence of an **XUID** entry in a pattern dictionary enables the PostScript interpreter to save cached instances of the pattern for later use, even when the pattern dictionary is loaded into virtual memory multiple times (for instance, by different jobs). To ensure correct behavior, **XUID** values must be assigned from a central registry. This is particularly appropriate for patterns treated as named resources. Patterns that are created dynamically by an application program should *not* contain **XUID** entries. |
| **PaintProc** | procedure | *(Required)* A PostScript procedure for painting the pattern cell. |
| **BBox** | array | *(Required)* An array of four numbers in the pattern coordinate system, giving the coordinates of the left, bottom, right, and top edges, respectively, of the pattern cell's bounding box. These boundaries are used to clip the pattern cell and to determine its size for caching. |
| **XStep** | number | *(Required)* The desired horizontal spacing between pattern cells, measured in the pattern coordinate system. |
| **YStep** | number | *(Required)* The desired vertical spacing between pattern cells, measured in the pattern coordinate system. Note that **XStep** and **YStep** may differ from the dimensions of the pattern cell implied by the **BBox** entry. This allows tiling with irregularly shaped figures. **XStep** and **YStep** may be either positive or negative, but not zero. |
| **PaintType** | integer | *(Required)* A code that determines how the color of the pattern cell is to be specified:<br>1 *Colored tiling pattern.* The **PaintProc** procedure itself specifies the colors used to paint the pattern cell. When the **PaintProc** procedure begins execution, the current color is the one that was in effect at the time the tiling pattern was instantiated with **makepattern**.<br>2 *Uncolored tiling pattern.* The **PaintProc** procedure does not specify any color information. Instead, the entire pattern cell is painted with a separately specified color each time the tiling pattern is used. Essen- |

| | | |
|---|---|---|
| | | tially, the **PaintProc** procedure describes a *stencil* through which the current color is to be poured. The **PaintProc** procedure must not invoke operators that specify colors or other color-related parameters in the graphics state; otherwise, an **undefined** error will occur (see Section 4.8.1, "Types of Color Space"). Use of the **imagemask** operator is permitted, however, since it does not specify any color information. |
| **TilingType** | integer | *(Required)* A code that controls adjustments to the spacing of tiles relative to the device pixel grid:<br>1 *Constant spacing.* Pattern cells are spaced consistently—that is, by a multiple of a device pixel. To achieve this, **makepattern** may need to distort the pattern cell slightly by making small adjustments to **XStep**, **YStep**, and the transformation matrix. The amount of distortion does not exceed 1 device pixel.<br>2 *No distortion.* The pattern cell is not distorted, but the spacing between pattern cells may vary by as much as 1 device pixel, both horizontally and vertically, when the tiling pattern is painted. This achieves the spacing requested by **XStep** and **YStep** *on average*, but not for each individual pattern cell.<br>3 *Constant spacing and faster tiling.* Pattern cells are spaced consistently as in tiling type 1, but with additional distortion permitted to enable a more efficient implementation. |
| **Implementation** | any | An additional entry inserted in the dictionary by the **makepattern** operator, containing information used by the interpreter to achieve proper tiling of the pattern. The type and value of this entry are implementation-dependent. |

The pattern cell is described in its own coordinate system, defined when the tiling pattern is instantiated from its prototype with the **makepattern** operator. This *pattern coordinate system* is formed by concatenating the operator's *matrix* operand with the current transformation matrix at the time of instantiation. The pattern dictionary's **XStep**, **YStep**, and **BBox** values are interpreted in the pattern coordinate system, and the **PaintProc** procedure is executed within that coordinate system.

The placement of pattern cells in the tiling is based on the location of one *key pattern cell*, which is then displaced by multiples of **XStep** and **YStep** to replicate the pattern. The origin of the key pattern cell coincides with the origin of the pattern coordinate system; the phase of the tiling can be controlled by the translation

components of the **makepattern** operator's *matrix* operand. Because the pattern coordinate system is locked into user space at the time of instantiation, properties of the pattern that depend on the coordinate system, such as the size of the pattern cell and the phase of the tiling in device space, are frozen at that time and are unaffected by subsequent changes in the CTM or other graphics state parameters.

### PaintProc Procedure

As described above in Section 4.9.1, "Using Patterns," the first step in painting with a pattern is to establish the pattern dictionary as the current color in the graphics state. In the case of a tiling pattern, subsequent painting operations will tile the painted areas with the pattern cell described in the dictionary. Whenever it needs to obtain the pattern cell, the interpreter does the following:

1. Invokes **gsave**
2. Installs the graphics state that was in effect at the time the tiling pattern was instantiated, with certain parameters altered as documented in the description of the **makepattern** operator in Chapter 8
3. Pushes the pattern dictionary on the operand stack
4. Executes the pattern's **PaintProc** procedure
5. Invokes **grestore**

The **PaintProc** procedure is expected to consume its dictionary operand and to use the information at hand to paint the pattern cell. It must obey certain guidelines to avoid disrupting the environment in which it is executed:

- It should not invoke any of the operators listed in Appendix G as unsuitable for use in encapsulated PostScript files.
- It should not invoke **showpage**, **copypage**, or any device setup operator.
- Except for removing its dictionary operand, it should leave the stacks unchanged.
- It should have no side effects beyond painting the pattern cell. It should not alter objects in virtual memory or anywhere else. Because of the effects of caching, the **PaintProc** procedure is called at unpredictable times and in unpredictable environments. It should depend only on information in the pattern dictionary and should produce the same effect every time it is called.

## Colored Tiling Patterns

A *colored tiling pattern* is one whose color is self-contained. In the course of painting the pattern cell, the **PaintProc** procedure explicitly sets the color of each graphical element it paints. A single pattern cell can contain elements that are painted different colors; it can also contain sampled grayscale or color images.

A **Pattern** color space representing a colored tiling pattern requires no additional parameters and can be specified with just the color space family name **Pattern**. The color space operand to **setcolorspace** can be either the name **Pattern** or a one-element array containing the name **Pattern**. A second parameter, the *underlying color space*—required as a second element of the array for uncolored tiling patterns—may optionally be included, but is ignored when using colored tiling patterns.

A color value operand to **setcolor** in such a color space has a single component, a pattern dictionary whose **PaintType** value is 1. Example 4.20 shows how to establish a colored tiling pattern as the current color, where *pattern* is a pattern dictionary of paint type 1.

**Example 4.20**

```
[/Pattern] setcolorspace                % Alternatively, /Pattern setcolorspace
pattern setcolor
```

Subsequent executions of painting operators, such as **fill**, **stroke**, **show**, and **imagemask**, will use the designated pattern to tile the areas to be painted.

***Note:*** *The **image** operator in its five-operand form and the **colorimage** operator use a predetermined color space (**DeviceGray**, **DeviceRGB**, or **DeviceCMYK**) for interpreting their color samples, regardless of the current color space. Setting a **Pattern** color space has no effect on these operators. The one-operand (dictionary) form of **image** is not allowed, since numeric color components are not meaningful in a **Pattern** color space. The **imagemask** operator is allowed, however, because the image samples do not represent colors, but rather designate places where the current color is to be painted.*

Example 4.21 defines a colored tiling pattern and then uses it to paint a rectangle and a character glyph; Figure 4.10 shows the results.

**Example 4.21**

```
<<    /PatternType 1                          % Tiling pattern
      /PaintType 1                            % Colored
      /TilingType 1
      /BBox [0 0 60 60]
      /XStep 60
      /YStep 60

      /star
         {  gsave                             % Private procedure used by PaintProc
               0 12 moveto
               4 {144 rotate 0 12 lineto} repeat
               closepath fill
            grestore
         } bind

      /PaintProc
         {  begin                             % Push pattern on dictionary stack
               0.3 setgray                    % Set color for dark gray stars
                  15 15 translate star
                  30 30 translate star
               0.7 setgray                    % Set color for light gray stars
                  -30 0 translate star
                  30 -30 translate star
            end
         } bind
>>                                            % End prototype pattern dictionary

matrix                                        % Identity matrix
makepattern                                   % Instantiate the pattern
/Star4 exch def

120 120 184 120 4 copy                        % Two copies of rectangle operands

/Pattern setcolorspace
Star4 setcolor rectfill                       % Fill rectangle with stars
0.0 setgray rectstroke                        % Stroke black outline

/Times-Roman 270 selectfont
160 100 translate
0.9 setgray 0 0 moveto (A) show               % Paint glyph with gray
Star4 setpattern 0 0 moveto (A) show          % Paint glyph with stars
```

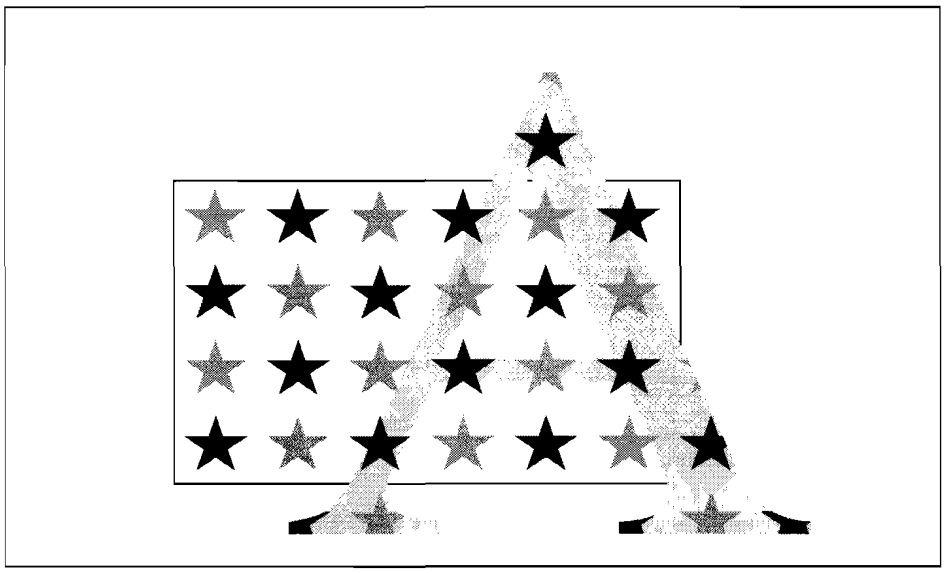


**FIGURE 4.10** *Output from Example 4.21*

The pattern consists of four stars in two different colors. The **PaintProc** procedure specifies the colors of the stars. Several features of Example 4.21 are noteworthy:

- After constructing the prototype pattern dictionary, the program immediately invokes **makepattern** on it. The value assigned to Star4 is the *instantiated* pattern returned by **makepattern**. There is no need to save the prototype pattern unless it is to be instantiated in multiple ways, perhaps with different sizes or orientations.
- The program illustrates both methods of selecting a pattern for painting. The first time, it invokes the **setcolorspace** and **setcolor** operators separately. The second time, it uses the convenience operator **setpattern**. Note that the calls to **setgray** also change the color space to **DeviceGray**.
- The rectangle and the glyph representing the letter A are painted with the same instantiated pattern (the pattern dictionary returned by a single execution of **makepattern**). The pattern cells align, even though the current transformation matrix is altered between the two uses of the pattern.
- The pattern cell does not completely cover the tile: it leaves the spaces between the stars unpainted. When the tiling pattern is used as a color, the existing

background shows through these unpainted areas, as the appearance of the A glyph in Figure 4.10 demonstrates. The letter is first painted solid gray; when it is painted again with the star pattern, the gray continues to show between the stars.

## Uncolored Tiling Patterns

An *uncolored tiling pattern* is one that has no inherent color: the color must be specified separately whenever the pattern is used. This provides a way to tile different regions of the page with pattern cells having the same shape but different colors. The pattern's **PaintProc** procedure does not explicitly specify any colors; it can use the **imagemask** operator, but not **image** or **colorimage**. (See Section 4.8.1, "Types of Color Space," for further discussion.)

A **Pattern** color space representing an uncolored tiling pattern requires a parameter: an array or name that identifies the *underlying color space* in which the actual color of the pattern is to be specified. Operands supplied to **setcolor** in such a color space must include both a color value in the underlying color space, specified by one or more numeric color components, and a pattern dictionary whose paint type is 2.

***Note:*** *The underlying color space of a* ***Pattern*** *color space cannot itself be a* ***Pattern*** *color space.*

Example 4.22 establishes an uncolored tiling pattern as the current color, using **DeviceRGB** as the underlying color space. The component values *r*, *g*, and *b* specify a color in **DeviceRGB** space; *pattern* is a pattern dictionary with a paint type of 2.

**Example 4.22**

```
[ /Pattern
    [/DeviceRGB]
] setcolorspace
r g b pattern setcolor
```

Subsequent executions of painting operators, such as **fill**, **stroke**, **show**, and **imagemask**, will tile the areas to be painted with the pattern cell defined by *pattern*, using the color specified by the components *r*, *g*, and *b*.

Example 4.23 defines an uncolored tiling pattern and then uses it to paint a rectangle and a circle in different colors; Figure 4.11 shows the results.

**Example 4.23**

```
<<    /PatternType 1                          % Tiling pattern
      /PaintType 2                            % Uncolored
      /TilingType 1
      /BBox [-12 -12 12 12]
      /XStep 30
      /YStep 30

      /PaintProc
         {  pop                               % Pop pattern dictionary
            0 12 moveto
            4{144 rotate 0 12 lineto} repeat
            closepath
            fill
         } bind
>>                                            % End prototype pattern dictionary

matrix                                        % Identity matrix
makepattern                                   % Instantiate the pattern
/Star exch def

140 110 170 100 4 copy                        % Two copies of rectangle operands

0.9 setgray rectfill                          % Fill rectangle with gray
[/Pattern /DeviceGray] setcolorspace
1.0 Star setcolor rectfill                    % Fill rectangle with white stars

225 185 60 0 360 arc                          % Build circular path
0.0 Star setpattern gsave fill grestore       % Fill circle with black stars
0.0 setgray stroke                            % Stroke black outline
```

The pattern consists of a single star, which the **PaintProc** procedure paints without first specifying a color. Most of the remarks after Example 4.21 on page 255 also apply to Example 4.23. Additionally:

- The program paints the rectangle twice, first with gray, then with the tiling pattern. To paint with the pattern, it supplies two operands to **setcolor**: the number 1.0, denoting white in the underlying **DeviceGray** color space, and the pattern dictionary.

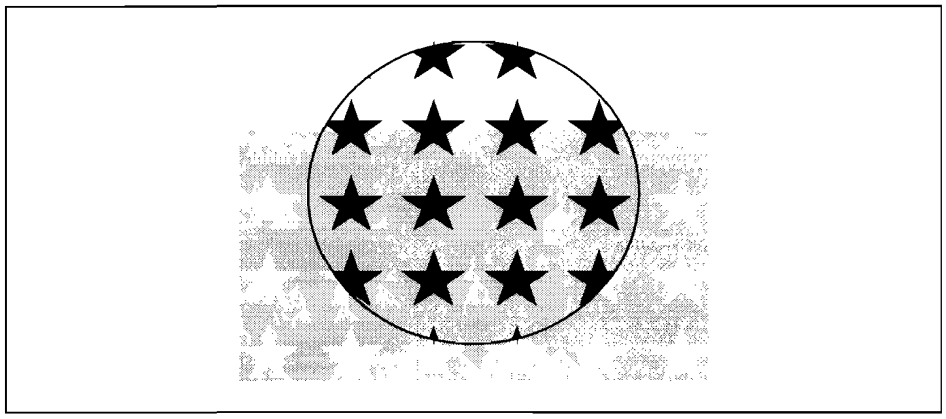


**FIGURE 4.11** *Output from Example 4.23*

- The program paints the circle with the same pattern, but with the color set to 0.0 (black). Note that in this instance, **setpattern** inherits parameters from the existing color space (see the description of the **setpattern** operator in Chapter 8 for details).

### 4.9.3 Shading Patterns

Shading patterns *(LanguageLevel 3)* provide a smooth transition between colors across an area to be painted, independent of the resolution of any particular output device and without specifying the number of steps in the color transition. Patterns of this type are described by pattern dictionaries with a pattern type of 2. Table 4.10 shows the contents of this type of dictionary, most of whose entries are identical to corresponding entries in the type 1 pattern dictionary (Table 4.9). The most significant entry is **Shading**, whose value is a *shading dictionary* defining the properties of the shading pattern's *gradient fill.* This is a complex "paint" that determines the type of color transition the shading pattern produces when painted across an area.

**TABLE 4.10 Entries in a type 2 pattern dictionary**

| KEY | TYPE | VALUE |
|---|---|---|
| **PatternType** | integer | *(Required)* A code identifying the pattern type that this dictionary describes; must be 2 for a shading pattern. |
| **Shading** | dictionary | *(Required)* A shading dictionary defining the shading pattern's gradient fill. The contents of the dictionary consist of the entries in Table 4.11 plus those in one of Tables 4.12 to 4.17. To ensure predictable behavior, once the pattern has been instantiated with the **makepattern** operator, this shading dictionary and all of its contents should be treated as if they were read-only. |
| **XUID** | array | *(Optional)* An *extended unique ID* that uniquely identifies the pattern (see Section 5.6.2, "Extended Unique ID Numbers). The presence of an **XUID** entry in a pattern dictionary enables the PostScript interpreter to save cached instances of the pattern for later use, even when the pattern dictionary is loaded into virtual memory multiple times (for instance, by different jobs). To ensure correct behavior, **XUID** values must be assigned from a central registry. This is particularly appropriate for patterns treated as named resources. Patterns that are created dynamically by an application program should *not* contain **XUID** entries. |
| **Implementation** | any | An additional entry inserted in the dictionary by the **makepattern** operator, containing information used by the interpreter to achieve proper shading of the pattern. The type and value of this entry are implementation-dependent. |

By setting a shading pattern as the current color in the graphics state, a PostScript program can use it with painting operators such as **fill**, **stroke**, **show**, or **imagemask** to paint a path, glyph, or mask with a smooth color transition. When a shading is used in this way, the geometry of the gradient fill is independent of the geometry of the object being painted.

When the area to be painted is a relatively simple shape whose geometry is the same as that of the gradient fill itself, the **shfill** operator can be used instead. **shfill** accepts a shading dictionary as an operand and applies the corresponding gradient fill directly to current user space. This operator does not require the creation of a pattern dictionary and works without reference to the current path or current color in the graphics state. See the description of the **shfill** operator in Chapter 8 for details.

***Note:*** *Patterns defined by type 2 pattern dictionaries do not tile. To create a tiling pattern containing a gradient fill, invoke the* ***shfill*** *operator from the* ***PaintProc*** *procedure of a type 1 (tiling) pattern.*

## Shading Dictionaries

A shading dictionary specifies details of a particular gradient fill, including the type of shading to be used, the geometry of the area to be shaded, and the geometry of the gradient fill itself. Various shading types are available, depending on the value of the dictionary's **ShadingType** entry:

- *Function-based shadings* (type 1) define the color of every point in the domain using a mathematical function (not necessarily smooth or continuous).
- *Axial shadings* (type 2) define a color blend along a line between two points, optionally extended beyond the boundary points by continuing the boundary colors.
- *Radial shadings* (type 3) define a blend between two circles, optionally extended beyond the boundary circles by continuing the boundary colors. This type of shading is commonly used to represent three-dimensional spheres and cones.
- *Free-form Gouraud-shaded triangle meshes* (type 4) define a common construct used by many three-dimensional applications to represent complex colored and shaded shapes. Vertices are specified in free-form geometry.
- *Lattice-form Gouraud-shaded triangle meshes* (type 5) are based on the same geometrical construct as type 4, but with vertices specified as a pseudorectangular lattice.
- *Coons patch meshes* (type 6) construct a shading from one or more color patches, each bounded by four Bézier curves.
- *Tensor-product patch meshes* (type 7) are similar to type 6, but with 16 control points in each patch instead of 12.

Table 4.11 shows the entries that all shading dictionaries share in common; entries specific to particular shading types are described in the relevant sections below.

***Note:*** *Many of the following descriptions refer to "the coordinate space into which the shading is painted." For shadings used with a type 2 pattern dictionary, this is the*

*pattern coordinate system established at the time the pattern is instantiated with* ***makepattern****. For shadings used directly with the* ***shfill*** *operator, it is the current user space.*

**TABLE 4.11 Entries common to all shading dictionaries**

| KEY | TYPE | VALUE |
|---|---|---|
| **ShadingType** | integer | *(Required)* The shading type:<br>1 Function-based shading<br>2 Axial shading<br>3 Radial shading<br>4 Free-form Gouraud-shaded triangle mesh<br>5 Lattice-form Gouraud-shaded triangle mesh<br>6 Coons patch mesh<br>7 Tensor-product patch mesh |
| **ColorSpace** | name or array | *(Required)* The color space in which color values are expressed. May be any device, CIE-based, or special color space except a **Pattern** space. All color values in the shading are interpreted relative to this color space. See "Color Space: Special Considerations" on page 263 for further information. |
| **Background** | array | *(Optional)* An array of color components appropriate to the color space, specifying a single background color value. If present, this color is used before any painting operation involving the shading, to fill the entire area to be painted. The effect is as if the painting operation were performed twice: first with the background color and then again with the shading. |
| **BBox** | array | *(Optional)* An array of four numbers giving the left, bottom, right, and top coordinates, respectively, of the shading's bounding box. The coordinates are interpreted in the coordinate space into which the shading is painted. If present, this bounding box is applied as a temporary clipping boundary when the shading is painted, in addition to the current clipping path and any other clipping boundaries in effect at that time. |
| **AntiAlias** | boolean | *(Optional)* A flag indicating whether to filter the shading function to prevent aliasing artifacts. The shading operators sample shading functions at a rate determined by the resolution of the output device. Aliasing can occur if the function is not smooth—that is, if it has a high spatial frequency relative to the sampling rate. Anti-aliasing can be computationally expensive and is usually unnecessary, since most shading functions are smooth enough, or are sampled at a high enough frequency, to avoid aliasing effects. This feature may not be implemented on some devices, in which case this flag is ignored. Default value: *false*. |

Some types of shading dictionary include a **DataSource** entry, whose value is an array, string, or file containing arbitrary amounts of descriptive data characterizing the shading's gradient fill. Since a shading pattern may access its shading dictionary multiple times, the descriptive data must be provided in reusable form. An array or string is reusable, but its length is subject to an implementation limit. A file can be of unlimited length, but only positionable files are reusable. In-line data obtained from **currentfile** is not reusable, and must be converted into reusable form by means of the **ReusableStreamDecode** filter. Nonreusable data sources may be used only with the **shfill** operator.

In addition, some shading dictionaries also include a function dictionary defining how colors vary across the area to be shaded. In such cases, the shading dictionary usually defines the geometry of the shading, while the function dictionary defines the color transitions across that geometry. The function dictionary is required for some types of shading and optional for others. Function dictionaries are described in detail in Section 3.10.1, "Function Dictionaries."

***Note:*** *Discontinuous color transitions, or those with high spatial frequency, may exhibit aliasing effects when painted at low effective resolutions.*

## Color Space: Special Considerations

Conceptually, a shading determines a color value for each individual point within the area to be painted. In practice, however, the shading may actually be used to compute color values only for some subset of the points in the target area, with the colors of the intervening points determined by interpolation between the ones computed. PostScript implementations are free to use this strategy as long as the interpolated color values approximate those defined by the shading to within the tolerance specified by the smoothness parameter in the graphics state (see the description of the **setsmoothness** operator in Chapter 8). The **ColorSpace** entry common to all shading dictionaries not only defines the color space in which the shading specifies its color values, but also determines the color space in which color interpolation is performed.

***Note:*** *Some shading types (4, 5, 6, and 7) perform interpolation on a parametric value supplied as* input *to the shading's color mapping function, as described in the relevant sections below. This form of interpolation is conceptually distinct from the interpolation described here, which operates on the* output *color values produced by the color mapping function and takes place within the shading's target color space.*

Gradient fills between colors defined by most shadings are implemented using a variety of interpolation algorithms, and these algorithms are sensitive to the characteristics of the color space. Linear interpolation, for example, may have observably different results when applied in CMYK color space than in the CIE 1976 L*a*b* color space, even if the starting and ending colors are perceptually identical. The difference arises because the two color spaces are not linear relative to each other. Shadings are rendered according to the following rules:

- If **ColorSpace** is a device color space different from the native color space of the output device, color values in the shading will be converted to the native color space using the standard conversion formulas described in Section 7.2, "Conversions among Device Color Spaces." To optimize performance, these conversions may take place at any time (either before or after any interpolation on the color values in the shading). Thus, shadings defined with device color spaces may have color gradient fills that are less accurate and somewhat device-dependent. (This does not apply to axial and radial shadings—shading types 2 and 3—because those shading types perform gradient fill calculations on a single variable and then convert to parametric colors.)
- If **ColorSpace** is a CIE-based color space, all gradient fill calculations will be performed in that space. Conversion to device colors will occur only after all interpolation calculations are performed. Thus, the color gradients will be device-independent for the colors generated at each point.
- If **ColorSpace** is a **Separation** or **DeviceN** color space and the specified colorants are supported, no color conversion calculations are needed. If the specified colorants are not supported (so that the color space's *alternativeSpace* parameter must be used), gradient fill calculations will be performed in the designated **Separation** or **DeviceN** color space before conversion to the alternative space. Thus, nonlinear *tintTransform* functions will be accommodated for the best possible representation of the shading.
- If **ColorSpace** is an **Indexed** color space, all color values specified in the shading will be immediately converted to the base color space. Depending on whether the base color space is a device or CIE-based space, gradient fill calculations will be performed as stated above. Interpolation never occurs in an **Indexed** color space, which is quantized and inappropriate for calculations that assume a continuous range of colors. For similar reasons, an **Indexed** color space is not allowed in any shading whose color values are generated by a function; this applies to any shading dictionary that contains a **Function** entry.

## Shading Types

In addition to the entries listed in Table 4.11, all shading dictionaries must have entries specific to the type of shading they represent, as indicated by the value of their **ShadingType** key. The following sections describe the available shading types and the dictionary entries specific to each.

### *Type 1 (Function-Based) Shadings*

In type 1 (function-based) shadings, the color of every point in the domain is defined by a specified mathematical function. The function is not necessarily smooth or continuous. This is the most general of the available shading types, and is useful for shadings that cannot be adequately described with any of the other types. In addition to the entries in Table 4.11, a type 1 shading dictionary includes the entries listed in Table 4.12.

***Note:*** *This type of shading may not be used with an* ***Indexed*** *color space.*

**TABLE 4.12 Additional entries specific to a type 1 shading dictionary**

| KEY | TYPE | VALUE |
|---|---|---|
| **Domain** | array | *(Optional)* An array of four numbers specifying the rectangular domain of coordinates over which the color function(s) are defined. Default value: [0 1 0 1]. |
| **Matrix** | array | *(Optional)* A transformation matrix mapping the coordinate space specified by the **Domain** entry into the coordinate space in which the shading is painted. For example, to map the domain rectangle [0 1 0 1] to a 1-inch square with lower-left corner at coordinates (100, 100) in default user space, the **Matrix** value would be [72 0 0 72 100 100]. Default value: the identity matrix [1 0 0 1 0 0]. |
| **Function** | dictionary or array | *(Required)* A 2-in, *n*-out function dictionary or an array of *n* 2-in, 1-out function dictionaries (where *n* is the number of color components in the shading dictionary's color space). Each function dictionary's domain must be a superset of that of the shading dictionary. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value. |

The domain rectangle (**Domain**) establishes an internal coordinate space for the shading that is independent of the coordinate space in which it is to be painted. The color function(s) (**Function**) specify the color of the shading at each point within this domain rectangle. The transformation matrix (**Matrix**) then maps the domain rectangle into a corresponding rectangle or parallelogram in the coordinate space in which the shading is painted. Points within the shading's bounding box (**BBox**) that fall outside this transformed domain rectangle will be painted with the shading's background color (**Background**); if the shading dictionary has no **Background** entry, such points will be left unpainted. If the function is undefined at any point within the declared domain rectangle, an **undefinedresult** error may occur, even if the corresponding transformed point falls outside the shading's bounding box.

### *Type 2 (Axial) Shadings*

Type 2 (axial) shadings define a color blend that varies along a linear axis between two endpoints and extends indefinitely perpendicular to that axis. The shading may optionally be extended beyond either or both endpoints by continuing the boundary colors indefinitely. In addition to the entries in Table 4.11 on page 262, a type 2 shading dictionary includes the entries listed in Table 4.13.

***Note:*** *This type of shading may not be used with an* ***Indexed*** *color space.*

**TABLE 4.13 Additional entries specific to a type 2 shading dictionary**

| KEY | TYPE | VALUE |
|---|---|---|
| **Coords** | array | *(Required)* An array of four numbers [$x_0$ $y_0$ $x_1$ $y_1$] specifying the starting and ending coordinates of the axis, expressed in the coordinate space in which the shading is painted. |
| **Domain** | array | *(Optional)* An array of two numbers [$t_0$ $t_1$] specifying the limiting values of a parametric variable $t$. The variable is considered to vary linearly between these two values as the color gradient varies between the starting and ending points of the axis. The variable $t$ becomes the argument with which the color function(s) are called. Default value: [0 1]. |
| **Function** | dictionary or array | *(Required)* A 1-in, $n$-out function dictionary or an array of $n$ 1-in, 1-out function dictionaries (where $n$ is the number of color components in the shading dictionary's color space). The function(s) are called with values of the parametric variable $t$ in the domain defined by the shading dictionary's **Domain** entry. Each function dictionary's domain must be a superset of that |

| | | |
|---|---|---|
| | | of the shading dictionary. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value. |
| **Extend** | array | *(Optional)* An array of two boolean values specifying whether to extend the shading beyond the starting and ending points of the axis, respectively. Default value: [*false false*]. |

The color blend is accomplished by linearly mapping each point $(x, y)$ along the axis between the endpoints $(x_0, y_0)$ and $(x_1, y_1)$ to a corresponding point in the domain specified by the shading dictionary's **Domain** entry. The point $(0, 0)$ in the domain corresponds to $(x_0, y_0)$ on the axis, and $(1, 0)$ corresponds to $(x_1, y_1)$. Since all points along a line in domain space perpendicular to the line from $(0, 0)$ to $(1, 0)$ will have the same color, only the new value of $x$ needs to be computed:

$$x' = \frac{(x_1 - x_0)(x - x_0) + (y_1 - y_0)(y - y_0)}{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

The value of the parametric variable $t$ is then determined from $x'$ as follows:

- For $0 \le x' \le 1$, $t = t_0 + (t_1 - t_0)x'$.
- For $x' < 0$, if the first value in the **Extend** array is *true*, then $t = t_0$; otherwise, $t$ is undefined and the point is left unpainted.
- For $x' > 1$, if the second value in the **Extend** array is *true*, then $t = t_1$; otherwise, $t$ is undefined and the point is left unpainted.

The value of $t$ is then passed as the input argument to the function(s) defined by the shading dictionary's **Function** entry, yielding the component values of the color with which to paint the point $(x, y)$.

### *Type 3 (Radial) Shadings*

Type 3 (radial) shadings define a color blend that varies between two circles. They are commonly used to depict three-dimensional spheres and cones. In addition to the entries in Table 4.11 on page 262, a type 3 shading dictionary includes the entries listed in Table 4.14.

***Note:*** *This type of shading may not be used with an* ***Indexed*** *color space.*

**TABLE 4.14 Additional entries specific to a type 3 shading dictionary**

| KEY | TYPE | VALUE |
|---|---|---|
| **Coords** | array | *(Required)* An array of six numbers [$x_0$ $y_0$ $r_0$ $x_1$ $y_1$ $r_1$] specifying the centers and radii of the starting and ending circles, expressed in the coordinate space in which the shading is painted. The radii $r_0$ and $r_1$ must both be greater than or equal to 0. If one radius is 0, the corresponding circle is treated as a point; if both are 0, nothing is painted. |
| **Domain** | array | *(Optional)* An array of two numbers [$t_0$ $t_1$] specifying the limiting values of a parametric variable $t$. The variable is considered to vary linearly between these two values as the color gradient varies between the starting and ending circles. The variable $t$ becomes the argument with which the color function(s) are called. Default value: [0 1]. |
| **Function** | dictionary or array | *(Required)* A 1-in, $n$-out function dictionary or an array of $n$ 1-in, 1-out function dictionaries (where $n$ is the number of color components in the shading dictionary's color space). The function(s) are called with values of the parametric variable $t$ in the domain defined by the shading dictionary's **Domain** entry. Each function dictionary's domain must be a superset of that of the shading dictionary. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value. |
| **Extend** | array | *(Optional)* An array of two boolean values specifying whether to extend the shading beyond the starting and ending circles, respectively. Default value: [*false false*]. |

The color blend is based on a family of *blend circles* interpolated between the starting and ending circles that are defined by the shading dictionary's **Coords** entry. The blend circles are defined in terms of a subsidiary parametric variable

$$s = \frac{t - t_0}{t_1 - t_0}$$

which varies linearly between 0.0 and 1.0 as $t$ varies across the domain from $t_0$ to $t_1$, as specified by the dictionary's **Domain** entry. The center and radius of each blend circle are given by the parametric equations

$$\begin{aligned} x_c(s) &= x_0 + s \times (x_1 - x_0) \\ y_c(s) &= y_0 + s \times (y_1 - y_0) \\ r(s) &= r_0 + s \times (r_1 - r_0) \end{aligned}$$

Each value of $s$ between 0.0 and 1.0 determines a corresponding value of $t$, which is then passed as the input argument to the function(s) defined by the shading dictionary's **Function** entry. This yields the component values of the color with which to fill the corresponding blend circle. For values of $s$ not lying between 0.0 and 1.0, the boolean values in the shading dictionary's **Extend** entry determine whether and how the shading will be extended. If the first of the two boolean values is *true*, the shading is extended beyond the defined starting circle to values of $s$ less than 0.0; if the second boolean value is *true*, the shading is extended beyond the defined ending circle to $s$ values greater than 1.0.

Note that either of the starting or ending circles may be larger than the other. If the shading is extended at the smaller end, the family of blend circles continues as far as that value of $s$ for which the radius of the blend circle $r(s) = 0$; if the shading is extended at the larger end, the blend circles continue as far as that $s$ value for which $r(s)$ is large enough to encompass the shading's entire bounding box (**BBox**). Extending the shading can thus cause painting to extend beyond the areas defined by the two circles themselves.

Conceptually, all of the blend circles are painted in order of increasing values of $s$, from smallest to largest. Blend circles extending beyond the starting circle are painted in the same color defined by the shading dictionary's **Function** entry for the starting circle ($s = 0.0$, $t = t_0$); those extending beyond the ending circle are painted in the color defined for the ending circle ($s = 1.0$, $t = t_1$). The painting is opaque, with the color of each circle completely overlaying those preceding it; thus if a point lies within more than one blend circle, its final color will be that of the last of the enclosing circles to be painted, corresponding to the greatest value of $s$. Note the following points:

- If one of the starting and ending circles entirely contains the other, the shading will depict a sphere.

- If neither circle contains the other, the shading will depict a cone. If the starting circle is larger, the cone will appear to point out of the page; if the ending circle is larger, the cone will appear to point into the page.

### *Type 4 Shadings (Free-Form Gouraud-Shaded Triangle Meshes)*

Type 4 shadings (free-form Gouraud-shaded triangle meshes) are commonly used to represent complex colored and shaded three-dimensional shapes. The area to be shaded is defined by a path composed entirely of triangles. The color at each vertex of the triangles is specified, and a technique known as Gouraud interpolation is used to color the interiors. The interpolation functions defining the shading may be linear or nonlinear. In addition to the entries in Table 4.11 on page 262, a type 4 shading dictionary includes the entries listed in Table 4.15.

**TABLE 4.15 Additional entries specific to a type 4 shading dictionary**

| KEY | TYPE | VALUE |
|---|---|---|
| **DataSource** | array, string, or file | *(Required)* The sequence of vertex coordinates and colors defining the free-form triangle mesh. |
| **BitsPerCoordinate** | integer | *(Required, unless **DataSource** is an array)* The number of bits used to represent each vertex coordinate. Allowed values are 1, 2, 4, 8, 12, 16, 24, and 32. |
| **BitsPerComponent** | integer | *(Required, unless **DataSource** is an array)* The number of bits used to represent each color component. Allowed values are 1, 2, 4, 8, 12, and 16. |
| **BitsPerFlag** | integer | *(Required, unless **DataSource** is an array)* The number of bits used to represent the edge flag for each vertex (see below). Allowed values of **BitsPerFlag** are 2, 4, and 8, but only the least significant 2 bits in each flag value are used. Allowed values for the edge flag itself are 0, 1, and 2. |
| **Decode** | array | *(Required, unless **DataSource** is an array)* An array of numbers describing how to map vertex coordinates and color components into the appropriate ranges of values. The decoding method is similar to that used in image dictionaries (see "Sample Decoding" on page 299). The ranges are specified as follows:<br>$[x_{\min}\ x_{\max}\ y_{\min}\ y_{\max}\ c_{1,\min}\ c_{1,\max} \cdots\ c_{n,\min}\ c_{n,\max}]$<br>Note that only one pair of $c$ values should be specified if a **Function** entry is present. |

| | | |
|---|---|---|
| **Function** | dictionary or array | *(Optional)* A 1-in, $n$-out function dictionary or an array of $n$ 1-in, 1-out function dictionaries (where $n$ is the number of color components in the shading dictionary's color space). If this entry is present, the color data for each vertex must be specified by a single parametric variable rather than by $n$ separate color components; the designated function(s) will be called with each interpolated value of the parametric variable to determine the actual color at each point. If **DataSource** is a string or a file, each input value will be clipped to the range interval specified for the corresponding color component in the shading dictionary's **Decode** array. If **DataSource** is an array (in which case **Decode** is not relevant), each input value will be clipped to the interval [0.0 1.0]. In either case, each function dictionary's domain must be a superset of the stated interval. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value.<br><br>This entry may not be used with an **Indexed** color space. |

The shading dictionary's **DataSource** entry provides a sequence of vertex coordinates and color data that defines the triangle mesh. Each vertex is specified by the following values, in the order shown:

$$f\ x\ y\ c_1 \ldots c_n$$

where

$f$ is the vertex's edge flag (discussed below)
$x$ and $y$ are its horizontal and vertical coordinates
$c_1 \ldots c_n$ are its color components

All vertex coordinates are expressed in the coordinate space in which the shading is painted. If the shading dictionary includes a **Function** entry, then only a single parametric value, $t$, is permitted for each vertex in place of the color components $c_1 \ldots c_n$.

The *edge flag* associated with each vertex determines the way it connects to the other vertices of the triangle mesh. A vertex $v_a$ with an edge flag value $f_a = 0$ begins a new triangle, unconnected to any other. At least two more vertices ($v_b$ and $v_c$) must be provided, but their edge flags will be ignored. These three vertices define a triangle ($v_a$, $v_b$, $v_c$), as shown in Figure 4.12.

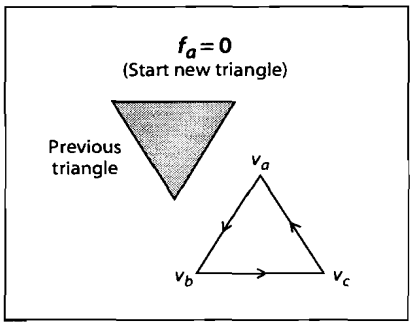



**FIGURE 4.12** *Starting a new triangle in a free-form Gouraud-shaded triangle mesh*

Subsequent triangles are defined by a single new vertex combined with two vertices of the preceding triangle. Given triangle $(v_a, v_b, v_c)$, where vertex $v_a$ precedes vertex $v_b$ in the data source and $v_b$ precedes $v_c$, a new vertex $v_d$ can form a new triangle on side $v_{bc}$ or side $v_{ac}$, as shown in Figure 4.13. (Side $v_{ab}$ is assumed to be shared with a preceding triangle and so is not available for continuing the mesh.) If the edge flag is $f_d = 1$ (side $v_{bc}$), the next vertex forms the triangle $(v_b, v_c, v_d)$; if the edge flag is $f_d = 2$ (side $v_{ac}$), the next vertex forms the triangle $(v_a, v_c, v_d)$. An edge flag of $f_d = 0$ would start a new triangle, as described above.

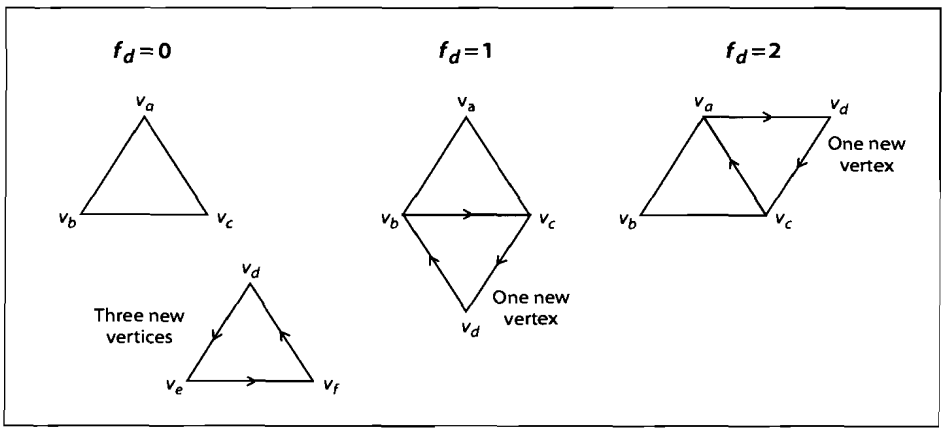



**FIGURE 4.13** *Connecting triangles in a free-form Gouraud-shaded triangle mesh*

Complex shapes can be created by using the edge flags to control the edge on which subsequent triangles are formed. Figure 4.14 shows two simple examples. Mesh 1 begins with triangle 1 and uses the following edge flags to draw each succeeding triangle:

1 ($f_a = f_b = f_c = 0$)
2 ($f_d = 1$)
3 ($f_e = 1$)
4 ($f_f = 1$)
5 ($f_g = 1$)
6 ($f_h = 1$)
7 ($f_i = 2$)
8 ($f_j = 2$)
9 ($f_k = 2$)
10 ($f_l = 1$)
11 ($f_m = 1$)

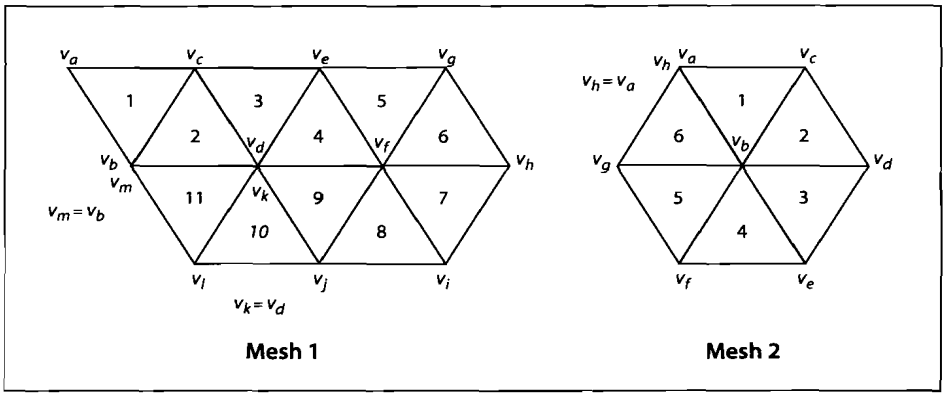



**FIGURE 4.14** *Varying the value of the edge flag to create different shapes*

Mesh 2 again begins with triangle 1 and uses the edge flags

1 ($f_a = f_b = f_c = 0$)
2 ($f_d = 1$)
3 ($f_e = 2$)
4 ($f_f = 2$)
5 ($f_g = 2$)
6 ($f_h = 2$)

The data source must provide vertex data for a whole number of triangles with appropriate edge flags; otherwise, a **rangecheck** error will occur. If the mesh contains only a few vertices, they may be represented by an array of numeric values (integers for edge flags, integers or real numbers for coordinates and colors); in

this case, only the **ShadingType**, **DataSource**, and **Function** entries in the shading dictionary are relevant. If the mesh contains many vertices, the data should be encoded compactly and drawn from a string or a file. The encoding is specified by the dictionary's **BitsPerFlag**, **BitsPerCoordinate**, **BitsPerComponent**, and **Decode** entries.

The data for each vertex consists of the following items, reading in sequence from higher-order to lower-order bit positions:

- An edge flag, expressed in **BitsPerFlag** bits
- A pair of horizontal and vertical coordinates, each expressed in **BitsPerCoordinate** bits
- A set of $n$ color components (where $n$ is the number of components in the shading's color space), each expressed in **BitsPerComponent** bits, in the order expected by the **setcolor** operator

Each set of vertex data must occupy a whole number of bytes; if the total number of bits required is not divisible by 8, the last data byte for each vertex is padded at the end with extra bits, which are ignored. The coordinates and color values are decoded according to the **Decode** array in the same way as in an image dictionary; see "Sample Decoding" on page 299 for details.

If the shading dictionary contains a **Function** entry, the color data for each vertex must be specified by a single parametric value $t$, rather than by $n$ separate color components. All linear interpolation within the triangle mesh is done using the $t$ values; after interpolation, the results are passed to the function(s) specified in the **Function** entry to determine the color of each point.

### *Type 5 Shadings (Lattice-Form Gouraud-Shaded Triangle Meshes)*

Type 5 shadings (lattice-form Gouraud-shaded triangle meshes) are similar to type 4, but instead of using free-form geometry, their vertices are arranged in a *pseudorectangular lattice*, which is topologically equivalent to a rectangular grid. The vertices are organized into rows, which need not be geometrically linear (see Figure 4.15). In addition to the entries in Table 4.11 on page 262, a type 5 shading dictionary includes the entries listed in Table 4.16.

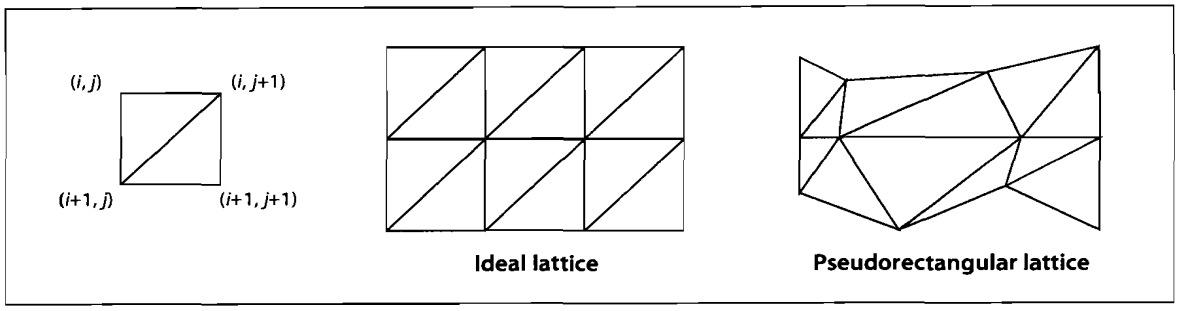



**FIGURE 4.15** *Lattice-form triangular meshes*

**TABLE 4.16 Additional entries specific to a type 5 shading dictionary**

| KEY | TYPE | VALUE |
|---|---|---|
| **DataSource** | array, string, or file | *(Required)* The sequence of vertex coordinates and colors defining the lattice-form triangle mesh. |
| **BitsPerCoordinate** | integer | *(Required, unless **DataSource** is an array)* The number of bits used to represent each vertex coordinate. Allowed values are 1, 2, 4, 8, 12, 16, 24, and 32. |
| **BitsPerComponent** | integer | *(Required, unless **DataSource** is an array)* The number of bits used to represent each color component. Allowed values are 1, 2, 4, 8, 12, and 16. |
| **VerticesPerRow** | integer | *(Required)* The number of vertices in each row of the lattice; must be greater than or equal to 2. The number of rows need not be specified. |
| **Decode** | array | *(Required, unless **DataSource** is an array)* An array of numbers describing how to map vertex coordinates and color components into the appropriate ranges of values. The decoding method is similar to that used in image dictionaries (see "Sample Decoding" on page 299). The ranges are specified as follows:<br>$[x_{\min}\ x_{\max}\ y_{\min}\ y_{\max}\ c_{1,\min}\ c_{1,\max} \cdots\ c_{n,\min}\ c_{n,\max}]$<br>Note that only one pair of $c$ values should be specified if a **Function** entry is present. |
| **Function** | dictionary or array | *(Optional)* A 1-in, $n$-out function dictionary or an array of $n$ 1-in, 1-out function dictionaries (where $n$ is the number of color components in the shading dictionary's color space). If this entry is present, the color data for each vertex must be specified by a single parametric variable rather than by $n$ separate color components; the designated function(s) will be called |

with each interpolated value of the parametric variable to determine the actual color at each point. If **DataSource** is a string or a file, each input value will be clipped to the range interval specified for the corresponding color component in the shading dictionary's **Decode** array. If **DataSource** is an array (in which case **Decode** is not relevant), each input value will be clipped to the interval [0.0 1.0]. In either case, each function dictionary's domain must be a superset of the stated interval. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value.

This entry may not be used with an **Indexed** color space.

---

The data source for a type 5 shading has the same format as for type 4, except that it does not use edge flags to define the geometry of the triangle mesh. The data for each vertex thus consists of the following values, in the order shown:

$x\ y\ c_1 \ldots c_n$

where

$x$ and $y$ are the vertex's horizontal and vertical coordinates
$c_1 \ldots c_n$ are its color components

All vertex coordinates are expressed in the coordinate space in which the shading is painted. If the shading dictionary includes a **Function** entry, then only a single parametric value, $t$, is permitted for each vertex in place of the color components $c_1 \ldots c_n$.

The **VerticesPerRow** entry in the shading dictionary gives the number of vertices in each row of the lattice. All of the vertices in a row are specified sequentially, followed by those for the next row. Given $m$ rows of $k$ vertices each, the triangles of the mesh are constructed using the following triplets of vertices, as shown in Figure 4.15:

$(V_{i,j}, V_{i,j+1}, V_{i+1,j})$ for $0 \le i \le m-2,\ 0 \le j \le k-2$
$(V_{i,j+1}, V_{i+1,j}, V_{i+1,j+1})$

See "Type 4 Shadings (Free-Form Gouraud-Shaded Triangle Meshes)" on page 270 for further details on the format of the vertex data.

### Type 6 Shadings (Coons Patch Meshes)

Type 6 shadings (Coons patch meshes) are constructed from one or more color patches, each bounded by four Bézier curves. Degenerate Bézier curves are allowed and are useful for certain graphical effects. At least one complete patch must be specified.

A Coons patch generally has two independent aspects:

- Colors are specified for each corner of the unit square, and bilinear interpolation is used to fill in colors over the entire unit square.
- Coordinates are mapped from the unit square into a four-sided patch whose sides are not necessarily linear. The mapping is continuous: the corners of the unit square map to corners of the patch, and the sides of the unit square map to sides of the patch, as shown in Figure 4.16.

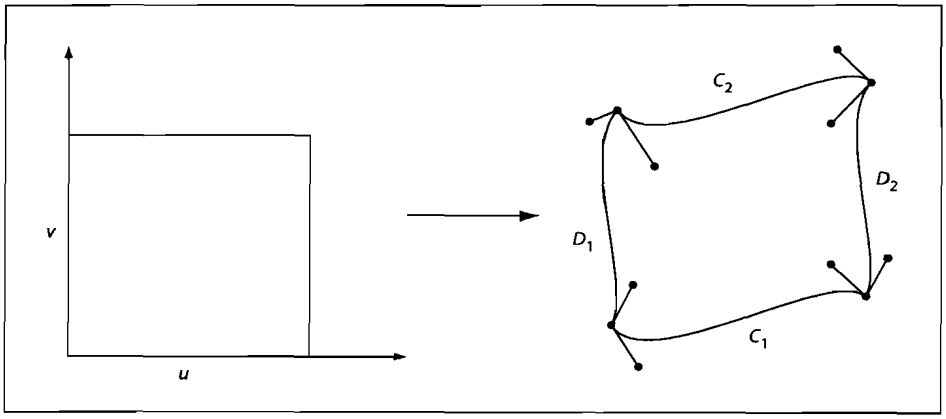



**FIGURE 4.16** *Coordinate mapping from a unit square to a four-sided Coons patch*

The sides of the patch are given by four cubic Bézier curves, $C_1$, $C_2$, $D_1$, and $D_2$, defined over a pair of parametric variables $u$ and $v$ that vary horizontally and ver-

tically across the unit square. The four corners of the Coons patch satisfy the equations

$$C_1(0) = D_1(0)$$
$$C_1(1) = D_2(0)$$
$$C_2(0) = D_1(1)$$
$$C_2(1) = D_2(1)$$

Two surfaces can be described that are linear interpolations between the boundary curves. Along the $u$ axis, the surface $S_C$ is defined by

$$S_C(u, v) = (1 - v) \times C_1(u) + v \times C_2(u)$$

Along the $v$ axis, the surface $S_D$ is given by

$$S_D(u, v) = (1 - u) \times D_1(v) + u \times D_2(v)$$

A third surface is the bilinear interpolation of the four corners:

$$S_B(u, v) = (1 - v) \times [(1 - u) \times C_1(0) + u \times C_1(1)] + v \times [(1 - u) \times C_2(0) + u \times C_2(1)]$$

The coordinate mapping for the shading is given by the surface $S$, defined as

$$S = S_C + S_D - S_B$$

This defines the geometry of each patch. A patch mesh is constructed from a sequence of one or more such colored patches.

Patches can sometimes appear to fold over on themselves—for example, if a boundary curve intersects itself. As the value of parameter $u$ or $v$ increases in parameter space, the location of the corresponding pixels in device space may change direction, so that new pixels are mapped onto previous pixels already mapped. If more than one point $(u, v)$ in parameter space is mapped to the same point in device space, the point selected will be the one with the largest value of $v$; if multiple points have the same $v$, the one with the largest value of $u$ will be selected. If one patch overlaps another, the patch that appears later in the data source paints over the earlier one.

Note also that the patch is a control surface, rather than a painting geometry. The outline of a projected square (that is, the painted area) may not be the same as the

patch boundary if, for example, the patch folds over on itself, as shown in Figure 4.17.

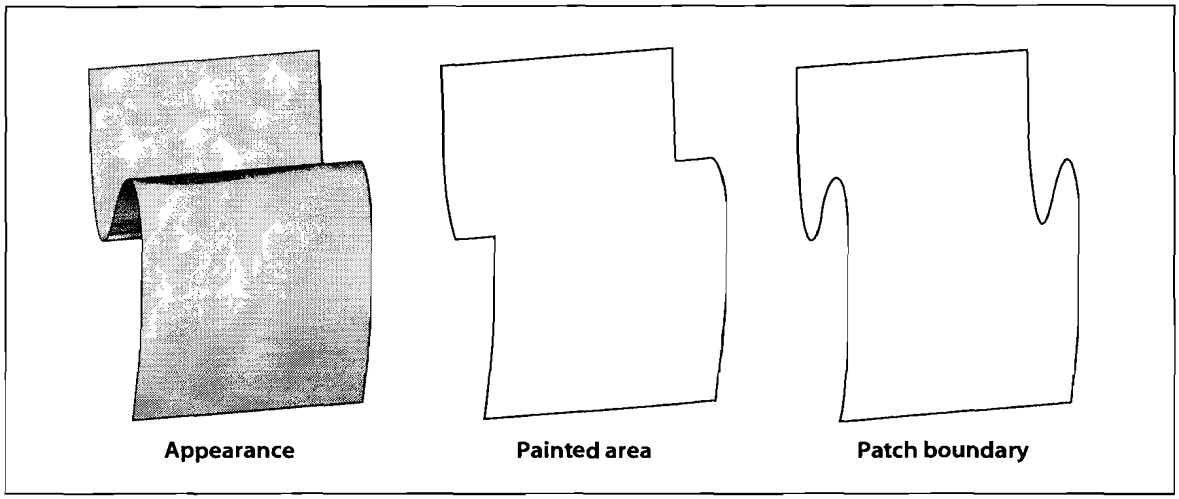



**FIGURE 4.17** *Painted area and boundary of a Coons patch*

In addition to the entries in Table 4.11 on page 262, a type 6 shading dictionary includes the entries listed in Table 4.17.

**TABLE 4.17 Additional entries specific to a type 6 shading dictionary**

| KEY | TYPE | VALUE |
|---|---|---|
| **DataSource** | array, string, or file | *(Required)* The sequence of coordinates and colors defining the patch mesh. |
| **BitsPerCoordinate** | integer | *(Required, unless **DataSource** is an array)* The number of bits used to represent each geometric coordinate. Allowed values are 1, 2, 4, 8, 12, 16, 24, and 32. |
| **BitsPerComponent** | integer | *(Required, unless **DataSource** is an array)* The number of bits used to represent each color component. Allowed values are 1, 2, 4, 8, 12, and 16. |
| **BitsPerFlag** | integer | *(Required, unless **DataSource** is an array)* The number of bits used to represent the edge flag for each patch (see below). Allowed values of **BitsPerFlag** are 2, 4, and 8, but only the least significant 2 bits in each flag value are used. Allowed values for the edge flag itself are 0, 1, 2, and 3. |

| | | |
|---|---|---|
| **Decode** | array | *(Required, unless **DataSource** is an array)* An array of numbers describing how to map coordinates and color components into the appropriate ranges of values. The decoding method is similar to that used in image dictionaries (see "Sample Decoding" on page 299). The ranges are specified as follows:<br>$[x_{min}\ x_{max}\ y_{min}\ y_{max}\ c_{1,min}\ c_{1,max} \cdots c_{n,min}\ c_{n,max}]$<br>Note that only one pair of $c$ values should be specified if a **Function** entry is present. |
| **Function** | dictionary or array | *(Optional)* A 1-in, $n$-out function dictionary or an array of $n$ 1-in, 1-out function dictionaries (where $n$ is the number of color components in the shading dictionary's color space). If this entry is present, the color data for each vertex must be specified by a single parametric variable rather than by $n$ separate color components; the designated function(s) will be called with each interpolated value of the parametric variable to determine the actual color at each point. If **DataSource** is a string or a file, each input value will be clipped to the range interval specified for the corresponding color component in the shading dictionary's **Decode** array. If **DataSource** is an array (in which case **Decode** is not relevant), each input value will be clipped to the interval [0.0 1.0]. In either case, each function dictionary's domain must be a superset of the stated interval. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value.<br>This entry may not be used with an **Indexed** color space. |

The dictionary's data source provides a sequence of Bézier control points and color values that define the shape and colors of each patch. All of a patch's control points are given first, followed by the color values for its corners. Note that this differs from a triangle mesh (shading types 4 and 5), in which the coordinates and color of each vertex are given together. All control point coordinates are expressed in the coordinate space in which the shading is painted.

As in triangle meshes, the data source can be either an array of numeric values or a string or stream containing encoded values, whose representation is specified by the **BitsPerFlag**, **BitsPerCoordinate**, **BitsPerComponent**, and **Decode** entries. In the latter case, if the total number of data bits required to define the patch is not divisible by 8, the last byte is padded at the end with extra bits, which are ignored.

As in free-form triangle meshes (type 4), each patch has an *edge flag* that tells which edge, if any, it shares with the previous patch. An edge flag of 0 begins a new patch, unconnected to any other. This must be followed by 12 pairs of coordinates, $x_1\ y_1\ x_2\ y_2\ \ldots\ x_{12}\ y_{12}$, which specify the Bézier control points that define the four boundary curves. Figure 4.18 shows how these control points correspond to the Bézier curves $C_1$, $C_2$, $D_1$, and $D_2$ identified in Figure 4.16 on page 277. Color values are then given for the four corners of the patch, in the same order as the control points corresponding to the corners. Thus, $c_1$ is the color at coordinates $(x_1, y_1)$, $c_2$ at $(x_4, y_4)$, $c_3$ at $(x_7, y_7)$, and $c_4$ at $(x_{10}, y_{10})$, as shown in the figure.

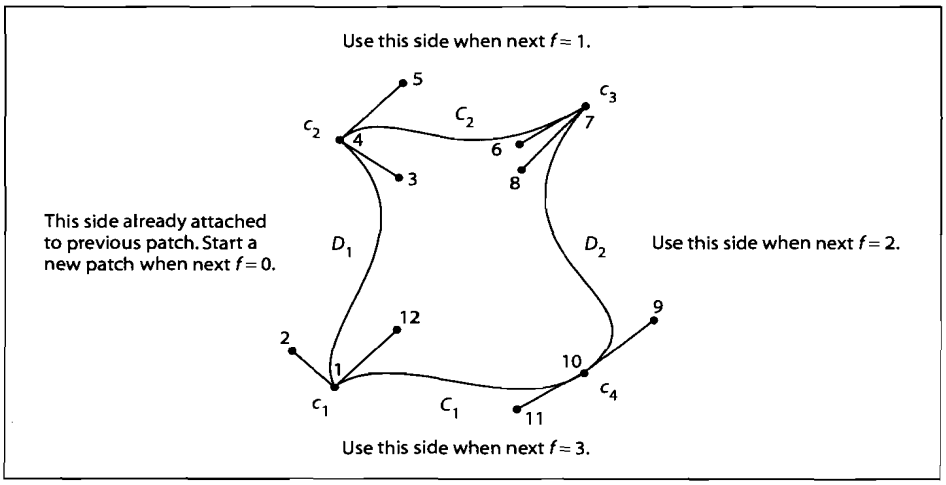



**FIGURE 4.18** *Color values and edge flags in Coons patch meshes*

Figure 4.18 also shows how nonzero values of the edge flag ($f = 1$, 2, or 3) connect a new patch to one of the edges of the previous patch. In this case, some of the previous patch's control points serve implicitly as control points for the new patch as well (see Figure 4.19), and so are not explicitly repeated in the data source. Table 4.18 summarizes the required data values for various values of the edge flag.

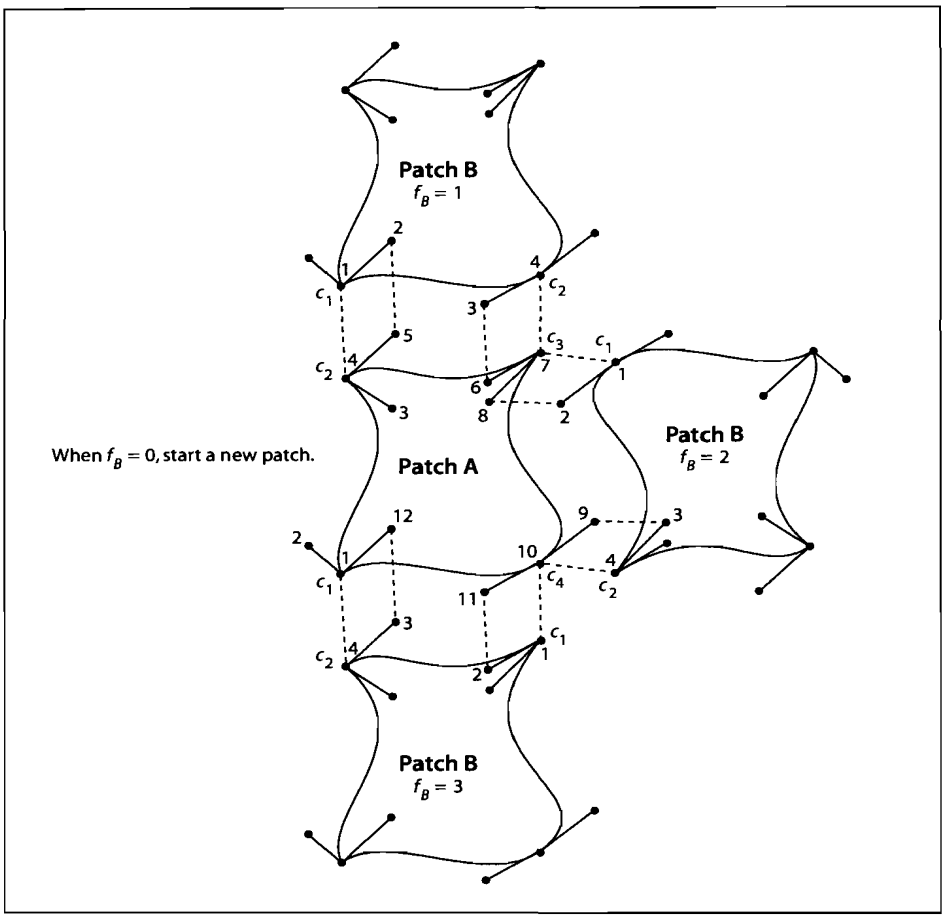



**FIGURE 4.19** *Edge connections in a Coons patch mesh*

**TABLE 4.18 Data values in a Coons patch mesh**

| EDGE FLAG | NEXT SET OF DATA VALUES |
|---|---|
| $f = 0$ | $x_1\ y_1\ x_2\ y_2\ x_3\ y_3\ x_4\ y_4\ x_5\ y_5\ x_6\ y_6\ x_7\ y_7\ x_8\ y_8\ x_9\ y_9\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}$ $c_1\ c_2\ c_3\ c_4$<br>New patch; no implicit values |

$f = 1$

$x_5\ y_5\ x_6\ y_6\ x_7\ y_7\ x_8\ y_8\ x_9\ y_9\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}$
$c_3\ c_4$

Implicit values:

$(x_1, y_1) = (x_4, y_4)$ previous
$(x_2, y_2) = (x_5, y_5)$ previous
$(x_3, y_3) = (x_6, y_6)$ previous
$(x_4, y_4) = (x_7, y_7)$ previous
$c_1 = c_2$ previous
$c_2 = c_3$ previous

$f = 2$

$x_5\ y_5\ x_6\ y_6\ x_7\ y_7\ x_8\ y_8\ x_9\ y_9\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}$
$c_3\ c_4$

Implicit values:

$(x_1, y_1) = (x_7, y_7)$ previous
$(x_2, y_2) = (x_8, y_8)$ previous
$(x_3, y_3) = (x_9, y_9)$ previous
$(x_4, y_4) = (x_{10}, y_{10})$ previous
$c_1 = c_3$ previous
$c_2 = c_4$ previous

$f = 3$

$x_5\ y_5\ x_6\ y_6\ x_7\ y_7\ x_8\ y_8\ x_9\ y_9\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}$
$c_3\ c_4$

Implicit values:

$(x_1, y_1) = (x_{10}, y_{10})$ previous
$(x_2, y_2) = (x_{11}, y_{11})$ previous
$(x_3, y_3) = (x_{12}, y_{12})$ previous
$(x_4, y_4) = (x_1, y_1)$ previous
$c_1 = c_4$ previous
$c_2 = c_1$ previous

If the shading dictionary contains a **Function** entry, the color data for each corner of a patch must be specified by a single parametric value $t$, rather than by $n$ separate color components $c_1 \ldots c_n$. All linear interpolation within the mesh is done using the $t$ values; after interpolation, the results are passed to the function(s) specified in the **Function** entry to determine the color of each point.

### *Type 7 Shadings (Tensor-Product Patch Meshes)*

Type 7 shadings (tensor-product patch meshes) are identical to type 6, except that they are based on a bicubic tensor-product patch defined by 16 control points, instead of the 12 control points that define a Coons patch. The shading dictionaries representing the two patch types differ only in the value of the **ShadingType** entry and in the number of control points specified for each patch

in the data source. Although the Coons patch is more concise and easier to use, the tensor-product patch affords greater control over color mapping.

Like the Coons patch mapping, the tensor-product patch mapping is controlled by the location and shape of four cubic Bézier curves marking the boundaries of the patch. However, the tensor-product patch has four additional, "internal" control points to adjust the mapping. The 16 control points can be arranged in a 4-by-4 array indexed by row and column, as follows (see Figure 4.20):

$$\begin{array}{cccc} p_{00} & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ p_{20} & p_{21} & p_{22} & p_{23} \\ p_{30} & p_{31} & p_{32} & p_{33} \end{array}$$

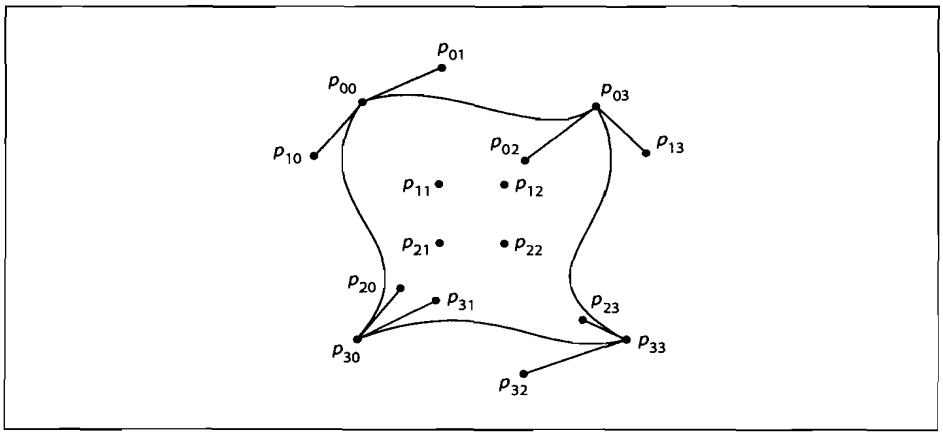



**FIGURE 4.20** *Control points in a tensor-product mesh*

As in a Coons patch mesh, the geometry of the tensor-product patch is described by a surface defined over a pair of parametric variables, $u$ and $v$, which vary horizontally and vertically across the unit square. The surface is defined by the equation

$$S(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} p_{ij} \times B_i(u) \times B_j(v)$$

where $p_{ij}$ is the control point in row $i$ and column $j$ of the tensor, and $B_i$ and $B_j$ are the *Bernstein polynomials*

$$B_0(t) = (1-t)^3$$
$$B_1(t) = 3t \times (1-t)^2$$
$$B_2(t) = 3t^2 \times (1-t)$$
$$B_3(t) = t^3$$

Since each point $p_{ij}$ is actually a pair of coordinates $(x_{ij}, y_{ij})$, the surface can also be expressed as

$$x(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} x_{ij} \times B_i(u) \times B_j(v)$$

$$y(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} y_{ij} \times B_i(u) \times B_j(v)$$

The geometry of the tensor-product patch can be visualized in terms of a cubic Bézier curve moving from the top boundary of the patch to the bottom. At the top and bottom, the control points of this curve coincide with those of the patch's top ($p_{00} \ldots p_{03}$) and bottom ($p_{30} \ldots p_{33}$) boundary curves, respectively. As the curve moves from the top edge of the patch to the bottom, each of its four control points follows a trajectory that is in turn a cubic Bézier curve defined by the four control points in the corresponding column of the array. That is, the starting point of the moving curve follows the trajectory defined by control points $p_{00} \ldots p_{30}$, the trajectory of the ending point is defined by points $p_{03} \ldots p_{33}$, and those of the two intermediate control points by $p_{01} \ldots p_{31}$ and $p_{02} \ldots p_{32}$. Equivalently, the patch can be considered to be traced by a Bézier curve moving from the left edge to the right, with its control points following the trajectories defined by the rows of the coordinate array instead of the columns.

The Coons patch (type 6) is actually a special case of the tensor-product patch (type 7) in which the four internal control points ($p_{11}, p_{12}, p_{21}, p_{22}$) are implicit-

ly defined by the boundary curves. The values of the internal control points are given by the equations

$$p_{11} = S(1/3, 2/3)$$
$$p_{12} = S(2/3, 2/3)$$
$$p_{21} = S(1/3, 1/3)$$
$$p_{22} = S(2/3, 1/3)$$

where $S$ is the Coons surface equation

$$S = S_C + S_D - S_B$$

discussed above under "Type 6 Shadings (Coons Patch Meshes)" on page 277. In the more general tensor-product patch, the values of these four points are unrestricted.

The coordinates of the control points in a tensor-product patch are actually stored in the shading dictionary's data source in the following order:

| | | | |
|---|---|---|---|
| 1 | 12 | 11 | 10 |
| 2 | 13 | 16 | 9 |
| 3 | 14 | 15 | 8 |
| 4 | 5 | 6 | 7 |

All control point coordinates are expressed in the coordinate space in which the shading is painted. These are followed by the color values for the four corners of the patch, in the same order as the corners themselves. If the patch's edge flag $f = 0$, all 16 control points and four corner colors must be explicitly specified in the data source; if $f = 1$, 2, or 3, the control points and colors for the patch's shared edge are implicitly understood to be the same as those along the specified edge of the previous patch, and are not repeated in the data source. Table 4.19 summarizes the data values for various values of the edge flag $f$, expressed in terms of the row and column indices used in Figure 4.20.

**TABLE 4.19 Data values in a tensor-product patch mesh**

| EDGE FLAG | NEXT SET OF DATA VALUES |
|---|---|
| $f = 0$ | $x_{00}\ y_{00}\ x_{10}\ y_{10}\ x_{20}\ y_{20}\ x_{30}\ y_{30}\ x_{31}\ y_{31}\ x_{32}\ y_{32}\ x_{33}\ y_{33}\ x_{23}\ y_{23}$<br>$x_{13}\ y_{13}\ x_{03}\ y_{03}\ x_{02}\ y_{02}\ x_{01}\ y_{01}\ x_{11}\ y_{11}\ x_{21}\ y_{21}\ x_{22}\ y_{22}\ x_{12}\ y_{12}$<br>$c_{00}\ c_{30}\ c_{33}\ c_{03}$<br>New patch; no implicit values |
| $f = 1$ | $x_{31}\ y_{31}\ x_{32}\ y_{32}\ x_{33}\ y_{33}\ x_{23}\ y_{23}\ x_{13}\ y_{13}\ x_{03}\ y_{03}$<br>$x_{02}\ y_{02}\ x_{01}\ y_{01}\ x_{11}\ y_{11}\ x_{21}\ y_{21}\ x_{22}\ y_{22}\ x_{12}\ y_{12}$<br>$c_{33}\ c_{03}$<br>Implicit values:<br>$(x_{00}, y_{00}) = (x_{30}, y_{30})$ previous; $c_{00} = c_{30}$ previous<br>$(x_{10}, y_{10}) = (x_{31}, y_{31})$ previous; $c_{30} = c_{33}$ previous<br>$(x_{20}, y_{20}) = (x_{32}, y_{32})$ previous<br>$(x_{30}, y_{30}) = (x_{33}, y_{33})$ previous |
| $f = 2$ | $x_{31}\ y_{31}\ x_{32}\ y_{32}\ x_{33}\ y_{33}\ x_{23}\ y_{23}\ x_{13}\ y_{13}\ x_{03}\ y_{03}$<br>$x_{02}\ y_{02}\ x_{01}\ y_{01}\ x_{11}\ y_{11}\ x_{21}\ y_{21}\ x_{22}\ y_{22}\ x_{12}\ y_{12}$<br>$c_{33}\ c_{03}$<br>Implicit values:<br>$(x_{00}, y_{00}) = (x_{33}, y_{33})$ previous; $c_{00} = c_{33}$ previous<br>$(x_{10}, y_{10}) = (x_{23}, y_{23})$ previous; $c_{30} = c_{03}$ previous<br>$(x_{20}, y_{20}) = (x_{13}, y_{13})$ previous<br>$(x_{30}, y_{30}) = (x_{03}, y_{03})$ previous |
| $f = 3$ | $x_{31}\ y_{31}\ x_{32}\ y_{32}\ x_{33}\ y_{33}\ x_{23}\ y_{23}\ x_{13}\ y_{13}\ x_{03}\ y_{03}$<br>$x_{02}\ y_{02}\ x_{01}\ y_{01}\ x_{11}\ y_{11}\ x_{21}\ y_{21}\ x_{22}\ y_{22}\ x_{12}\ y_{12}$<br>$c_{33}\ c_{03}$<br>Implicit values:<br>$(x_{00}, y_{00}) = (x_{03}, y_{03})$ previous; $c_{00} = c_{03}$ previous<br>$(x_{10}, y_{10}) = (x_{02}, y_{02})$ previous; $c_{30} = c_{00}$ previous<br>$(x_{20}, y_{20}) = (x_{01}, y_{01})$ previous<br>$(x_{30}, y_{30}) = (x_{00}, y_{00})$ previous |